**### capture script name & command line args**

**SCRIPT\_NAME=$0**

**SCRIPT\_BASE\_NAME=$(basename "${SCRIPT\_NAME}")**

**SCRIPT\_ULL\_NAME=$(readlink -f "${SCRIPT\_NAME}") ### Expand full path**

**SCRIPT\_PATH=$(dirname "${SCRIPT\_FULL\_NAME}")**

**COMMAND\_LINE="${SCRIPT\_FULL\_NAME}$\*"**

1.SCRIPT\_NAME=$0

This captures the name of the script being executed.

2.Base Name:

SCRIPT\_BASE\_NAME=$(basename "${SCRIPT\_NAME}")

This extracts the base name of the script, stripping any leading directory path.

3.Full Path:

SCRIPT\_FULL\_NAME=$(readlink -f "${SCRIPT\_NAME}")

This resolves the full, absolute path of the script, even following symlinks.

4.Expand Full Path:

SCRIPT\_PATH=$(dirname "${SCRIPT\_FULL\_NAME}")

This extracts the directory path from the full script name.

5.Command Line Arguments:

COMMAND\_LINE="${SCRIPT\_FULL\_NAME} $\*"

This combines the full script name with all command-line arguments passed to the script.

Essentially, these lines help your script understand its own location and how it was invoked, which is useful for creating portable, self-aware scripts.

**source "${SCRIPT\_PATH}/fhir\_common\_functions.sh"**

* source: Executes the commands from a specified file in the current shell.
* ${SCRIPT\_PATH}: This variable contains the path to the directory where your script is located.
* fhir\_common\_functions.sh: This is the name of the script containing common functions you want to include in your current script.

When you use this line in your script, it essentially imports the functions and variables from fhir\_common\_functions.sh, making them available in your current script. This is useful for reusing code and keeping your scripts modular.

**Function vshow() {**

**while (( $# )); do**

**local param="$1"**

**if [[ "${param%/%}" != "${param}" ]]; then**

**param=$(compgen -v | grep "^${param%/%}")**

**fi**

**shift**

**local i**

**for i in $param; do**

**if [[ "${i}" =~ [Pp][Aa][Ss][Ss] || "${i}" =~ [Pp][Ss][Ww][Dd] ]]; then**

**echo "## $i=\*\*\*\*\*\*\*\*"**

**else**

**typeset spaces="##"**

**local indent=""**

**echo "$spaces $i=${!i}"**

**fi**

**done**

**done**

**}**

In this script:

* while (( $# )): Loops through all arguments.
* local param="$1": Captures the current parameter.
* compgen -v | grep "^${param%/%}": Lists matching variables.
* if [[ "${i}" =~ [Pp][Aa][Ss][Ss] || "${i}" =~ [Pp][Ss][Ww][Dd] ]]: Checks if variable names contain "PASS" or "PWD", then masks them.
* echo "$spaces $i=${!i}": Displays variable values.

This setup helps secure sensitive information while displaying other variables.

**local x="$(eval echo "\#\# \"$(echo "$i=\${$i-#UNDEFINED#}")\"")"**

**while reade**

**do**

**if (( ! ${#indent} ))**

**then**

**echo "$REPLY"**

**indent="${spaces:0:$(expr index "$REPLY" "=")}**

**indent="${indent% }="**

**else**

**echo"${indent}$REPLY**

**fi;**

**done <<< "$(echo "$x")"**

**fi;**

**done**

**shift l;**

**done**

**}**

This part of the script is performing some variable evaluation and formatting. Here's a breakdown of what each piece does:

**Breakdown**

**1.Evaluate and Format Variables:**

**local x="$(eval echo "\#\# \"$(echo "$i=\${$i-#UNDEFINED#}")\"")"**

eval echo "\#\# \"$(echo "$i=\${$i-#UNDEFINED#}")\"")": Evaluates the value of the variable i, and if i is undefined, it uses #UNDEFINED# as the default value. The eval command executes the constructed command string.

2.**Reading Line by Line:**

while read -r; do

Starts a loop to read lines from the variable x.

3.Initial Indent:

if (( ! ${#indent} )); then

echo "$REPLY"

indent="${spaces:0:$(expr index "$REPLY" "=")}"

indent="${indent% }="

else

echo "${indent}$REPLY"

fi

Checks if indent is empty.

If true, it prints the first line of x.

Sets indent based on the position of the first = character in REPLY, ensuring alignment.

Otherwise, it prepends indent to subsequent lines.

4.Process the Text:

done <<< "$(echo "$x")"

Feeds the evaluated and formatted variable x into the while loop for processing.

This part of the script essentially processes and formats variables for consistent, aligned output.

explain dirs\_to\_make ="error processed processed/archive processed/result processed/result/archive processed/smile processed/smile/error processed/smile/processed processed/smile/result"

res\_to\_mark ="eob-ip-clinical eob-op-clinical eob-prof-clinical"

clients\_to\_make ="him hmde hmwv bcbsmn bcbsnd bcbswy"

envs\_to\_make ="tenv2 tenv3 tenvb tenvc"

1.**dirs\_to\_make:** A list of directories you plan to create.

dirs\_to\_make="error processed processed/archive processed/result processed/result/archive processed/smile processed/smile/error processed/smile/processed processed/smile/result"

2.**res\_to\_mark:** A list of resources to mark, probably for some specific processing tasks.

res\_to\_mark="eob-ip-clinical eob-op-clinical eob-prof-clinical"

**3.clients\_to\_make:** A list of client directories or identifiers.

clients\_to\_make="him hmde hmwv bcbsmn bcbsnd bcbswy"

4.**envs\_to\_make:** Environments you want to set up.

envs\_to\_make="tenv2 tenv3 tenvb tenvc"

This setup organizes your script by defining all necessary items at the beginning, which can then be looped over for creation or processing. Super-efficient for managing bulk tasks. Ready to dive into script loops to process these lists?

processed\_file\_ct=0

successful\_file\_ct=0

failed\_file\_ct=0

total\_records\_loadded=0

exit\_code=0

try\_eob\_module=0

do\_not\_call\_etl\_module=0;

processed\_file\_ct=0: Initializes a counter for the number of processed files.

successful\_file\_ct=0: Initializes a counter for the number of files processed successfully.

failed\_file\_ct=0: Initializes a counter for the number of files that failed to process.

total\_records\_loaded=0: Initializes a counter for the total number of records loaded.

exit\_code=0: Initializes an exit code, which usually indicates the success or failure of the script.

try\_eob\_module=0: Initializes a flag to determine if the EOB module should be tried.

do\_not\_call\_etl\_module=0: Initializes a flag to determine if the ETL module should not be called.

**function f\_write () {**

**echo -e "$(date +'%m/%d/%y %H:%M:%S') PID:[$$] ${\*}"**

**}**

Breakdown:-- date +'%m/%d/%y %H:%M:%S': This part gets the current date and time in the format MM/DD/YY HH:MM:SS.

**$$:** This special variable represents the process ID of the current shell.

**${}\*:** This stands for all the arguments passed to the function.

What It Does

When you call f\_write, it prints a log message to the terminal that includes the current date and time, the process ID, and the message you provide. The -e flag enables interpretation of backslash escapes.

**function f\_convert\_-bad\_chars\_to\_space () {**

**local file=$1**

**local tempfileExt="PID\_$$\_noNulls"**

**### check if file has any <null>s**

**local nullCt=$(grep -pan '\000|\032' $file |WC -1)**

**if ["$nullCt" -gt "0" ]; then**

**write "changing Bad Characters to spaces from file [$file] ---($nullCt) instances found"**

**### Change <null>s to space (OCTAL Char values)**

**tr '\000\032' ' ' <"${file}" > "${file} "${tempFleExt}" || exi**

**t\_error $? "Error removing <nulls> from file [$file]"**

**### Rename the new file W/O nullsto the original file name**

**mv -f "${file}.${tempfilEext}" "$file"**

**fi**

**}**

Function Breakdown

1. Define Function: ex:-- function f\_convert\_bad\_chars\_to\_space () {
2. Local Variables:

local file=$1

local tempfileExt="PID\_$$\_noNulls"

1. Check for Null Characters:

local nullCt=$(grep -pan '\000|\032' "$file" | wc -l)

if [ "$nullCt" -gt 0 ]; then

f\_write "Changing bad characters to spaces from file [$file] --- ($nullCt) instances found"

1. Replace Null Characters:

tr '\000\032' ' ' <"${file}" > "${file}.${tempfileExt}" || exit\_error $? "Error removing <nulls> from file [$file]"

1. Rename Temporary File:

mv -f "${file}.${tempfileExt}" "$file"

fi

Full Function with Proper Syntax and Formatting:---

function f\_convert\_bad\_chars\_to\_space () {

local file=$1

local tempfileExt="PID\_$$\_noNulls"

# Check if file has any <null>s

local nullCt=$(grep -pan '\000|\032' "$file" | wc -l)

if [ "$nullCt" -gt 0 ]; then

f\_write "Changing bad characters to spaces from file [$file] --- ($nullCt) instances found"

# Change <null>s to space (OCTAL Char values)

tr '\000\032' ' ' <"${file}" > "${file}.${tempfileExt}" || exit\_error $? "Error removing <nulls> from file [$file]"

# Rename the new file without nulls to the original file name

mv -f "${file}.${tempfileExt}" "$file"

fi

}

This function replaces null characters (octal values \000 and \032) in a file with spaces, logs the change, and updates the file. Efficiently keeps your files clean. Anything else you’re curious about?